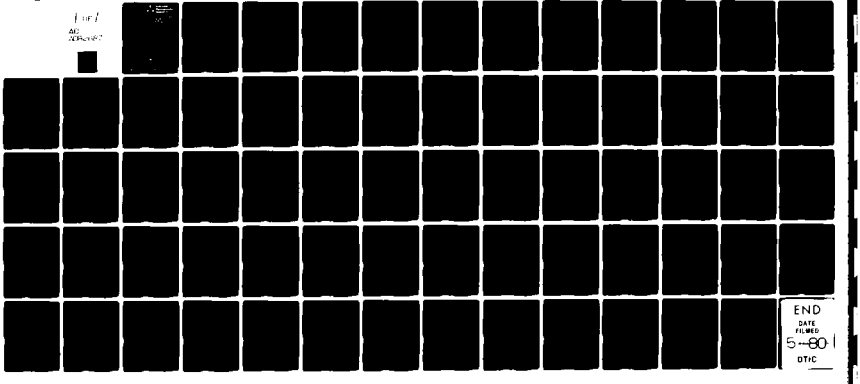
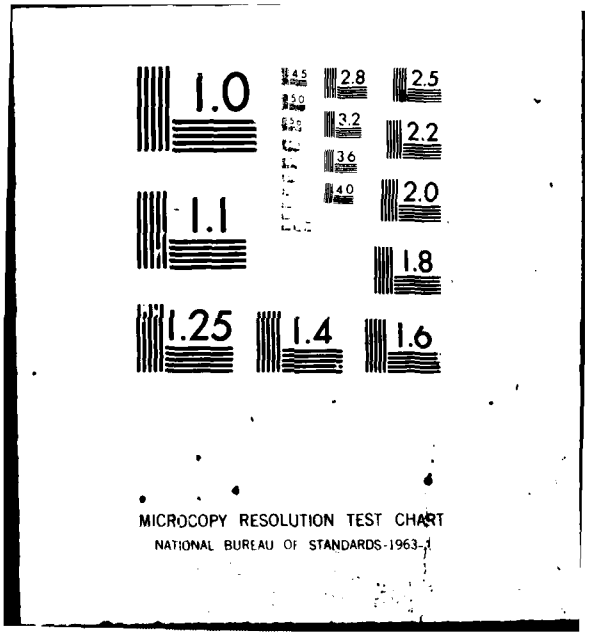
AD-A082 687 GENERAL ELECTRIC CO ARLINGTON VA F/G 9/2
EXPERIMENTAL EVALUATION OF ON-LINE PROGRAM CONSTRUCTION.(U)
DEC 79 S B SHEPPARD, P MILLIMAN, B CURTIS N00014-77-C-0158
UNCLASSIFIED TR-79-388100-6 NL



END
DATE FILMED
5-80
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Software Management Research

LEVEL II

AD A082687

EXPERIMENTAL EVALUATION OF ON-LINE PROGRAM CONSTRUCTION

SYLVIA B. SHEPPARD
PHIL MILLIMAN
BILL CURTIS

TR-79-388100-6
DECEMBER 1979

GENERAL ELECTRIC
INFORMATION SYSTEMS PROGRAMS
ARLINGTON, VIRGINIA

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)

Experimental Evaluation of On-line Program Construction

5. TYPE OF REPORT & PERIOD COVERED

Technical Report, Mar 78 - Feb 79
3/78 - 2/79

6. PERFORMING ORG. REPORT NUMBER

TR-79-388100-6

7. AUTHOR(s)

Sylvia B. Sheppard, Phil Milliman, Bill Curtis

8. CONTRACT OR GRANT NUMBER(s)

N00014-77-C-0158

9. PERFORMING ORGANIZATION NAME AND ADDRESS

General Electric Company, Suite 200
1755 Jefferson Davis Highway
Arlington, VA 22202

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

NR 197-037

11. CONTROLLING OFFICE NAME AND ADDRESS

Engineering Psychology Programs, Code 455
Office of Naval Research
Arlington, VA 22217

12. REPORT DATE

December 1979

13. NUMBER OF PAGES

83

14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)

Same

15. SECURITY CLASS. (of this report)

Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited. Reproduction in whole or in part is permitted for any purpose of the United States Government.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Same

18. SUPPLEMENTARY NOTES

Dr. John O'Hare, Assistant Director
Engineering Psychology Programs
Office of Naval Research

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software development; software complexity; software science; interactive programming.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

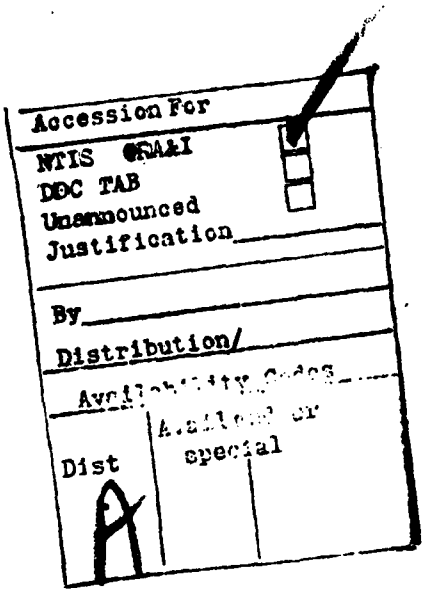
An experiment was conducted to assess the utility of complexity metrics for the prediction of programmer performance in the construction of software. After practicing on a preliminary program, each of the nine participants developed three experimental programs on-line. An English language description of each problem was presented in addition to one of the following specification formats: ~~1)~~ program design language, ~~2)~~ tree chart, and ~~3)~~ both of these techniques. ~~No significant differences were found in~~

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

the times to construct programs from these different types of specification formats. The software complexity metrics developed by Halstead and McCabe were found to be significantly better predictors of the time to complete the program than the number of statements.



TR-79-388100-6

EXPERIMENTAL EVALUATION OF ON-LINE PROGRAM CONSTRUCTION

by

Sylvia B. Sheppard,
Phil Milliman,
and
Bill Curtis

Software Management Research
Information Systems Programs
General Electric Company
1755 Jefferson Davis Highway
Arlington, VA 22202

Submitted to:

Office of Naval Research
Engineering Psychology Programs
Arlington, VA 22217

Contract: #N00014-77-C-0158
Work Unit: NR 197-037

December 1979

SOFTWARE COMPLEXITY RESEARCH PROGRAM

Department of Defense (DOD) software production and maintenance is a large, poorly understood, and inefficient process. Recently Frost and Sullivan (The Military Software Market, 1977) estimated the yearly cost for software within DOD to be as large as $9 billion. De Roze (1977) has also estimated that 115 major defense systems depend on software for their success. In an effort to find near-term solutions to software related problems, the DOD has begun to support research into the software production process. A formal 5 year R&D plan (Carlson & DeRoze, 1977) related to the management and control of computer resources was recently written in response to DOD Directive 5000.29(1976). This plan requested research leading to the identification and validation of metrics for software quality.

Interest continues to grow in the use of quantitative metrics which assess the complexity of software. Such metrics are assumed to be valuable aids in determining the quality of software. Boehm, Brown, and Lipow (1976) and McCall, Richards, and Walters (1977) have proposed combinations of such metrics which assess numerous factors that collectively constitute this nebulous "software quality". Such factors include reliability, portability, maintainability, and myriad other xxx-abilities.

There are numerous potential uses for measures which assess these various quality factors. First, they can be used as feedback to programmers during development, indicating potential problems with code they have developed (Elshoff, 1978). Use of metrics in this way would require guidelines for altering code so as to bring different metric values within acceptable limits.

A second use for metrics is in guiding software testing. McCabe (1976) proposed the cyclomatic number as a means of assessing the computational complexity of the software testing problem. Other metrics which index the quality or complexity of software may help identify modules or subroutines which are likely to be the most error-prone.

Another use for software metrics is their use in estimating maintenance requirements. If one or more metrics can be empirically related to the difficulty programmers experience in working with software, then more accurate estimates can be made of the staffing levels that will be necessary during maintenance. Empirical validity studies will be necessary before employing metrics for any of the

three uses described here. Such research should be conducted with professional programmers.

The experimental investigation described in this report is part of a research program seeking to provide information about the psychological and human resource aspects of computer programming. The challenge undertaken in this research was to quantify the psychological complexity of software. It is important to distinguish clearly between the psychological and computational complexity of software. Computational complexity refers to characteristics of algorithms or programs which make their proof of correctness difficult, lengthy, or impossible. For example, as the number of distinct paths through a program increases, the computational complexity also increases. Psychological complexity refers to those characteristics of software which make human understanding of software more difficult. No direct linear relationship between computational and psychological complexity is expected. A program with many control paths may not be psychologically complex. Any regularity to the branching process within a program may be used by a programmer to simplify understanding of the program.

Halstead (1977) has recently developed a theory concerned with the psychological aspects of computer programming. His theory provides objective estimates of the effort and time required to generate a program, the effort required to understand a program, and the number of bugs in a particular program (Fitzsimmons & Love, 1978). Some predictions of the theory are counterintuitive and contradict results of previous psychological research. The theory has attracted attention because independent tests of hypotheses derived from it have proven amazingly accurate.

Although predictions of programmer behavior have been particularly impressive, much of the research testing Halstead's theory has been performed without sufficient experimental or statistical controls. Further, much of the data were based upon imprecise estimating techniques. Nevertheless, the available evidence has been sufficient to justify a rigorous evaluation of the theory.

Rather than conduct a research program designed specifically to test Halstead's theory of software science, a research strategy was chosen which would generate suggestions for improving programmer efficiency regardless of the success of any particular theory. This research focused on four phases of the software life-cycle: understanding, modification, debugging, and construction. Since different cognitive processes are assumed to predominate in each phase,

no single experiment or set of experiments on a particular phase were believed to provide a sufficient basis for making broad recommendations for improving programmer efficiency. Each experiment in this research program was designed to test important variables assumed to affect a particular phase of software development. Professional programmers were used in these experiments to provide the greatest possible external validity for the results (Campbell & Stanley, 1966). In addition, the theory of software science and other related metrics were evaluated with these data. This experiment, the fourth in the series, concentrated on the construction phase of software development.

ABSTRACT

An experiment was conducted to assess the utility of complexity metrics for the prediction of programmer performance in the construction of software. After practicing on a preliminary program, each of the nine participants developed three experimental programs on-line. An English language description of each problem was presented in addition to one of the following specification formats: 1) program design language, 2) tree chart, and 3) both of these techniques. No significant differences were found in the times to construct programs from these different types of specification formats. The software complexity metrics developed by Halstead and McCabe were found to be significantly better predictors of the time to complete the program than the number of statements.

TABLE OF CONTENTS

INTRODUCTION

The impact of the cost of software has become increasingly apparent over the last decade. In the late 1950's most of the cost of computer systems was for hardware, but now 90% of the costs are for software (Shneiderman, 1980). The design and construction process has great impact on the subsequent operations and maintenance portions of a software system which account for most life cycle costs (Boehm 1973). Several experiments evaluating program construction have been performed (Boies & Gould, 1974; Dunsmore & Gannon, 1978; Love, 1977; Lucas & Kaplan, 1974; Miller & Thomas, 1976; Newsted, 1974; Shneiderman & Mayer, 1979; Shneiderman, Mayer, McKay, & Heller, 1977; Sime, Arblaster, & Green, 1977). One problem with these studies has been the inability to examine individual programmer strategies. Love (1977) and Dunsmore & Gannon (1978) were forced to deduce strategies from subjective examination of successive runs of programs, as were Sackman, Erickson, & Grant (1968), Sime et al. (1977) & Youngs (1974). Myers (1978) was only able to look at the final product.

In an attempt to collect more objective data about the programming process, the Software Management Research Unit at General Electric has established a software research laboratory. The microcomputer at the core of this lab keeps an audit trail of all the actions of a user during the construction, editing and debugging of a program. We can examine the time spent on various portions of tasks and actual changes and additions made, rather than relying on assumptions about programmer behavior. A description of the laboratory can be found in Appendix F.

The present experiment had three main purposes: 1) to demonstrate that the laboratory is a feasible data collection tool, 2) to examine the impact of two design specification formats, and 3) to assess various metrics of program complexity for predicting programming effort.

There are two primary approaches to design specification formats in current software projects: verbal and graphical descriptions (Jones, 1979). A number of studies in cognitive psychology have indicated that verbal descriptions are sequential in nature, emphasizing ordering relationships (Kintsch & van Dijk, 1978; Paivio, 1971). Wright and Reid (1973) have demonstrated that verbal descriptions are retained better over a period of days than those presented graphically.

The alternate approach to verbal specification of design is a graphic representation of the program. This method is well suited to stepwise refinement (Wirth, 1971) and to some theories of the organization of human memory (Collins and Quillian, 1969; Kintsch & van Dijk, 1978; Paivio, 1971; Ramsey, Atwood, & Van Doren, 1978). Although graph-oriented documentation is widely used throughout industry, flowcharting is one graphical method that has been shown to be of questionable utility (Newsted, 1979; Ramsey, Atwood, & Van Doren, 1978; Shneiderman, Mayer, McKay & Heller, 1977).

To compare the two forms of design, the two representations of a program must have the same information content. A number of authors have demonstrated mappings from trees or hierarchical structures to sequential constrained language such as a Program Design Language (Jackson, 1975; McClure, 1978; Stay, 1976). A tree allows a functional representation of a program, but it is difficult to represent the ordering and selection criteria. Jackson approaches this goal in his data trees with symbols for selection and iteration. In the present experiment a program is represented as a tree or a hierarchy of functions, with lower levels representing more detail. If there is some selection criteria, the edges of the graph are labeled with the criteria.

A Program Design Language (PDL) was chosen as the verbal sequential representation in this experiment. The PDL was chosen because it was possible to map precisely from the tree structure to the PDL.

A program specification represented in a tree structure must be translated into a sequential, program-like form. Since this format adds an additional translation step into the construction process, it can be argued that it is best to specify the detailed design with a verbal description originally. This study attempted to determine whether the specification format significantly influenced the construction process.

Previous experiments in this program of research (Curtis, Sheppard, & Milliman, 1979; Curtis, Sheppard, Milliman, Borst, & Love, 1979) have shown that the psychological complexity of a computer program can be quantified by using the Halstead (1977) and McCabe (1976) complexity metrics. These experiments were concerned with performance during comprehension, modification, and debugging tasks. Because the metrics are also relevant to the construction process, their ability to predict programming time was evaluated during this experiment.

Halstead's theory of software science argues that algorithms have measurable characteristics and that a number of useful measures can be derived from simple counts of operators and operands. From these quantities Halstead (1977) developed measures for the overall program length, potential smallest volume of an algorithm, actual volume of an algorithm (the difficulty of understanding a program), language level (a constant for a given language), programming effort (number of mental discriminations required to generate a program), program development time, and number of delivered bugs in a system. Halstead's theory has been the subject of considerable evaluative research (Fitzsimmons & Love, 1978). Correlations often greater than .90 have been reported between the Halstead metrics and such measures as the number of bugs in a program, programming time, debugging time, and algorithm purity.

Thomas McCabe (1976) has defined complexity in relation to the decision structure of a program. He assesses complexity as it affects the testability and reliability of a module. McCabe's complexity metric, $v(G)$, is the classical graph-theory cyclomatic number indicating the number of regions in a graph, or in the current usage, the number of linearly independent control paths comprising a program. As is true of Halstead's measures, McCabe's metric has been shown to be a better predictor of performance than a simple count of lines of code (Curtis, Sheppard, & Milliman, 1979). The present experiment sought to confirm these predictions in a program construction task.

METHOD

Participants

Nine professional programmers participated in this experiment. They averaged 4.7 years of programming experience, ranging from less than 1 to 12 years (SD = 4.1).

Procedure

A packet of materials prepared for each participant included: written instructions on the experimental procedure (Appendix A), instructions for using the operating system and the Fortran compiler (Appendix B), a short preliminary task (Appendix C), and three experimental tasks (Appendix D). Since the material in Appendix B was rather long, it was presented to the participants the day before the experiment so they could become familiar with the instructions prior to the experimental session.

A session was conducted with an individual programmer at the CRT terminal of a microcomputer. In addition to the written materials, the participants were given some prompting on-line in the form of instructions such as "Please turn to the first problem". Following an initial practice problem, participants were presented with three separate programs comprising their experimental tasks. A questionnaire was presented on-line after the experimental tasks were completed (Appendix E). The experimenter was present at all times and ready to act as a reference source for explaining how to use the computer, the editor, or the Fortran compiler.

Participants worked at their own speed, signaling the instructor when prepared to execute a compilation. Due to idiosyncracies in the automated data collection system, all programs were compiled and run by the experimenter. A program that was not correct at the first submission was returned to the participant, and repeated submissions were executed until the program had been run successfully. Successful completion required producing the correct output from an input data file that was hidden from the participant.

A detailed record of each response by a participant was recorded automatically by the data collection system of the microcomputer. An internal timer accurate to one-hundredth of a second recorded the time for each of these responses.

Independent Variables

Programs. Four short algorithms were selected for the general understandability of their content. The practice problem required computing the average of a list of numbers.

One experimental program required the alphabetic matching of strings of characters. Another required summing the positive and negative values of a set of numbers. The last algorithm found the maximum and minimum of a set of numbers. Because participants were expected to complete a practice problem and three experimental tasks within three hours, the algorithms were necessarily short and uncomplicated.

Documentation. The practice problem included a functional specification in natural language, sample input, and sample output. No additional documentation was given in the practice problem. The experimental tasks included one of three types of additional documentation: a PDL description of the program function, a tree structure showing the function, or both the PDL and tree structures.

The purpose of the additional documentation was to present the functional decomposition of the algorithm in detail. The PDL specification was a sequential, constrained language description of the functions to be performed in the program. Indentation was used to specify the more detailed levels of the process, thus making the description partly hierarchical in nature. The tree representation was designed for ordinary preorder traversal (Knuth, 1973). That is, the vertical dimension indicated levels of abstraction of the functions to be performed, with the most detailed levels occurring lowest in the tree. The horizontal dimension indicated order of progression from left to right with each "father" node being processed first. The leftmost "son" followed the "father", and the progession followed to the right until all "sons" had been processed. Branching and iteration were indicated by labelling the condition for execution on the edge leading to the node. Where no label appeared, the node was always executed.

Experimental Design

A within-subjects 3^2 factorial design was employed. Each of the three programs was prepared with the three types of documentation: the PDL, the tree structure or both the PDL and tree structures. A matrix of these nine experimental conditions is shown in Figure 1. Previous experiments employing the design had shown learning effects (Sheppard, Curtis, Milliman, & Love, 1979). Therefore, the order of presentation of the tasks was counterbalanced so that each program appeared as the first, second, or third task equally often. Each type of documentation was similarly counterbalanced according to order of presentation.

Individual Difference Measures

Scores on the practice problem were used as a measure of programming ability related to the experimental task. Participants were also asked to complete a questionnaire about their programming experience.

Dependent Variables

The major dependent variable was the total time required to successfully accomplish a task. An internal timer accurate to one one-hundredth of a second recorded each of the actions of the participant at the microprocessor. The total time was computed by summing all of these intervals. It did not include time for compilation and execution of the program, tasks performed by the experimenter.

A second dependent variable, the study time, was the time between the presentation of the problem to the participant and the initial entry of an instruction into the computer system (i.e., the first ADD command). This variable measured the planning or thinking time of the participant. The study time was included in the total time.

Complexity Metrics

Halstead's V and E. Halstead's volume (V) and effort (E) metrics were computed precisely from a program (based on Ottenstein, 1976) whose input was the source code listings of the 27 programs, three from each of the nine participants.

	DOCUMENTATION		
PROGRAM	PDL	TREE	PDL & TREE
1 ALPHABETIC MATCH	1, 4, 7	2, 5, 8	3, 6, 9
2 SUM POS. & NEG.VALUES	3, 6, 9	1, 4, 7	2, 5, 8
3 MAXIMUM & MINIMUM	2, 5, 8	3, 6, 9	1, 4, 7

Figure 1. Assignments of the 9 participants in the experimental design

The computational formulas are:

$$V = (N_1 + N_2) \log_2 (\eta_1 + \eta_2)$$

$$E = \frac{\eta_1 N_2 (N_1 + N_2) \log_2 (\eta_1 + \eta_2)}{2\eta_2}$$

where,

η_1 = # of unique operators

η_2 = # of unique operands

N_1 = total # of all operators

N_2 = total # of all operands

McCabe's v(G). McCabe's metric is the classical graph-theory cyclomatic number defined as:

v(G) = # edges - # nodes + 2(# connected components).

McCabe presents two simpler methods of calculating v(G): the number of predicate nodes plus 1, or the number of regions computed from a planar graph of the control flow.

Number of statements. The length of the program was the total number of statements added minus the number of statements deleted.

RESULTS

All 27 programs (nine participants with three programs each) were completed successfully. The number of tries to get a clean compilation ranged from 1 to 3 (M = 1.3), and the total tries to run successfully ranged from 1 to 4 (M = 1.7). The average total time to complete a program was 21.3 minutes (SD = 14.6), and the average study time was 3.3 minutes (SD = 4.2). There were significant differences in the times required to construct each of the three programs ($p \leq .001$), accounting for over 50% of the variance in performance. Program 1 required an average of 35.9 minutes to complete, while Programs 2 and 3 averaged 14.4 and 13.6 minutes, respectively.

Four metrics of program complexity were computed for each of the programs constructed by the participants: Halstead's V and E, McCabe's $v(G)$, and the number of program statements. Descriptive statistics for the metrics on each of the three programs are presented in Table 1. The mean values for Programs 2 (summing positive and negative numbers) and 3 (finding maximum and minimum values) were similar. The means for Program 1 (alphabetic matching) were larger, and the standard deviations for the Halstead and McCabe metrics were much greater. The range of individual differences in the implementation of these programs was striking, and these differences were most prominent on Halstead's effort metric.

Table 2 shows the intercorrelations among the software complexity metrics. As expected, V and E were highly correlated. Number of statements was not as well correlated with other metrics.

Pearson Product-Moment correlations between the metrics and both the total time and study time are reported in Table 3. Study times were unrelated to the metrics. However, over all 27 programs the Halstead and McCabe metrics were better predictors of total time when compared to the number of statements in the programs. The volume metric (r = .78) was a significantly ($p \leq .01$) better predictor of total time than was the effort metric (r = .61).

Table 1

Descriptive Statistics for Metrics by Program

Metric	Mean	Standard deviation	Range		Max/Min
			Minimum	Maximum	
Program 1:					
Statements	26.6	10.2	17	51	3.0
Halstead's $\underline{V}$	417.2	170.3	223	747	3.3
Halstead's $\underline{E}$	8816.1	8165.4	1951	27655	14.2
McCabe's $\underline{v(G)}$	14.4	9.3	6	34	5.7
Program 2:					
Statements	20.7	7.3	12	33	2.8
Halstead's $\underline{V}$	215.6	46.1	156	299	1.9
Halstead's $\underline{E}$	2523.5	118.5	1768	5199	2.9
McCabe's $\underline{v(G)}$	4.2	1.0	3	6	2.0
Program 3:					
Statements	19.1	10.9	12	46	3.8
Halstead's $\underline{V}$	188.4	87.5	126	404	3.2
Halstead's $\underline{E}$	2311.2	2766.0	908	9538	10.5
McCabe's $\underline{v(G)}$	4.9	0.6	4	6	1.5

Table 2

Intercorrelations Among Complexity Metrics

	Number of Statements	Halstead's Volume	Halstead's Effort
Halstead's V	.59***		
Halstead's E	.65***	.95***	
McCabe's v(G)	.52**	.81***	.85***

Note: $n = 27$
**$p \leq .01$
***$p \leq .001$

When the data were separated by program, strong differences emerged in the ability of the metrics to predict performance time. For Programs 2 and 3, both Halstead metrics were exceptionally strong predictors of time. No such relationship could be found for Program 1. Regressions for each metric on total time are presented in Figures 2 through 5 with scores for Program 1 circled in each case. Times for Program 1 were generally longer and showed more variability than those for Programs 2 and 3. The scatterplots in Figure 3 through 5 appear to support a curvilinear relationship. However, this appearance is almost entirely the result of two datapoints from Program 1.

Halstead (1977) presents a way of estimating the time required to generate a program. He develops this estimator by dividing the effort metric (which is presented as the number of mental discriminations required to generate the program) by the Stroud (1967) number of 18 mental discriminations per second. It is evident from Figure 4 that this measure consistently underestimates the actual amount of time required to develop the program.

No significant effects on performance were observed among the three methods of documentation that were presented with the programs.

The average pretest time was 30.9 minutes with an average of 3.1 tries to complete the program successfully.

Table 3

Correlations Between Performance Time and Software Complexity Metrics

Measures	Correlations			
	Number of statements	Halstead's Volume	Halstead's Effort	McCabe's v(G)
All Programs:				
Study Time	.30	.29	.21	.04
Total Time	.35*	.78***	.61***	.66***
Program 1:				
Study Time	.13	.02	-.01	-.37
Total Time	.15	.42	.28	.38
Programs 2 & 3:				
Study Time	.39	.04	.08	.07
Total Time	.20	.85***	.83***	.59**

Note: For all programs, $n = 27$; for Program 1, $n = 9$; for Programs 2 & 3, $n = 18$.

*$p \leq .05$
**$p \leq .01$
***$p \leq .001$

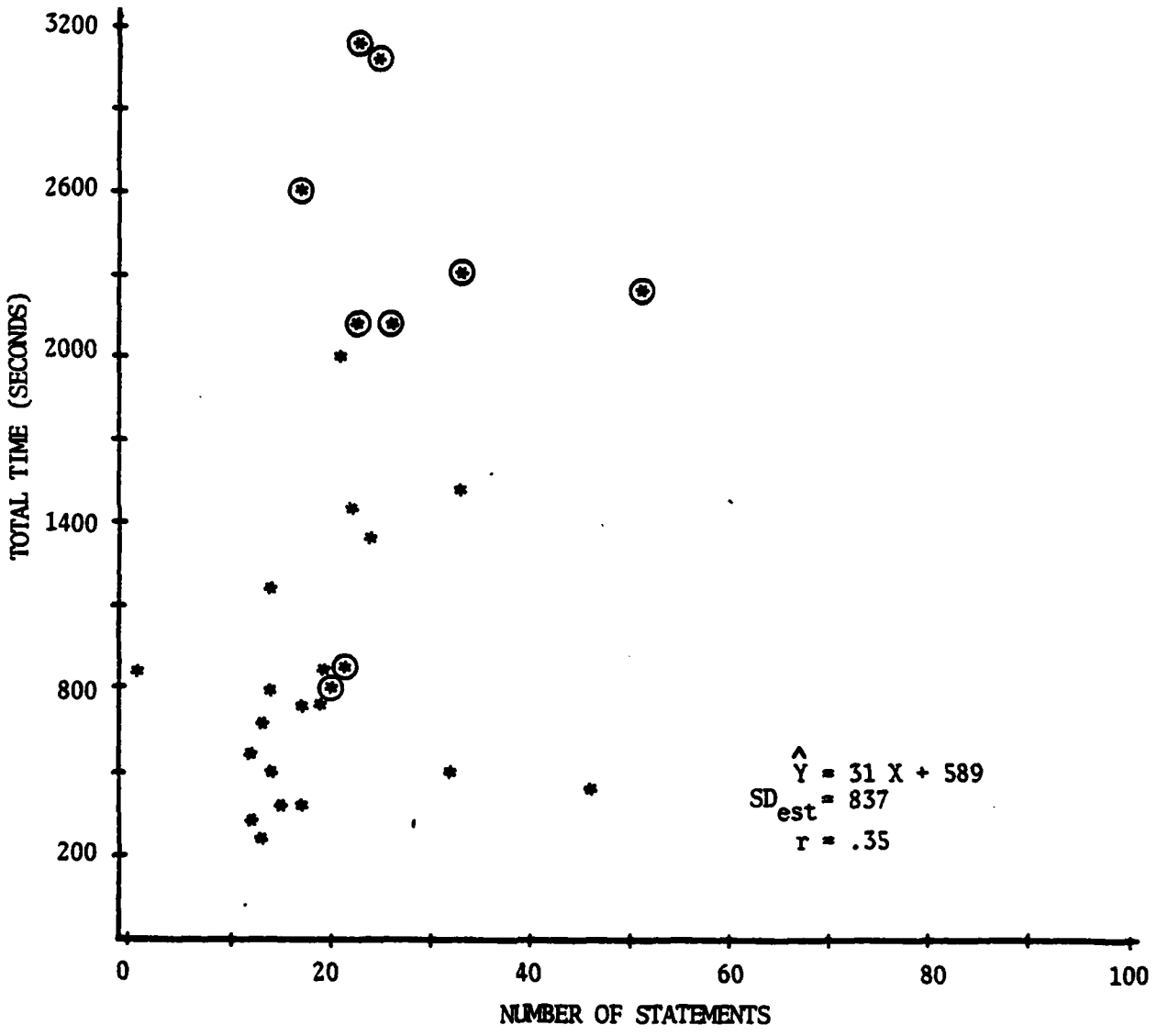


Figure 2. Scatterplot of number of statements with total time

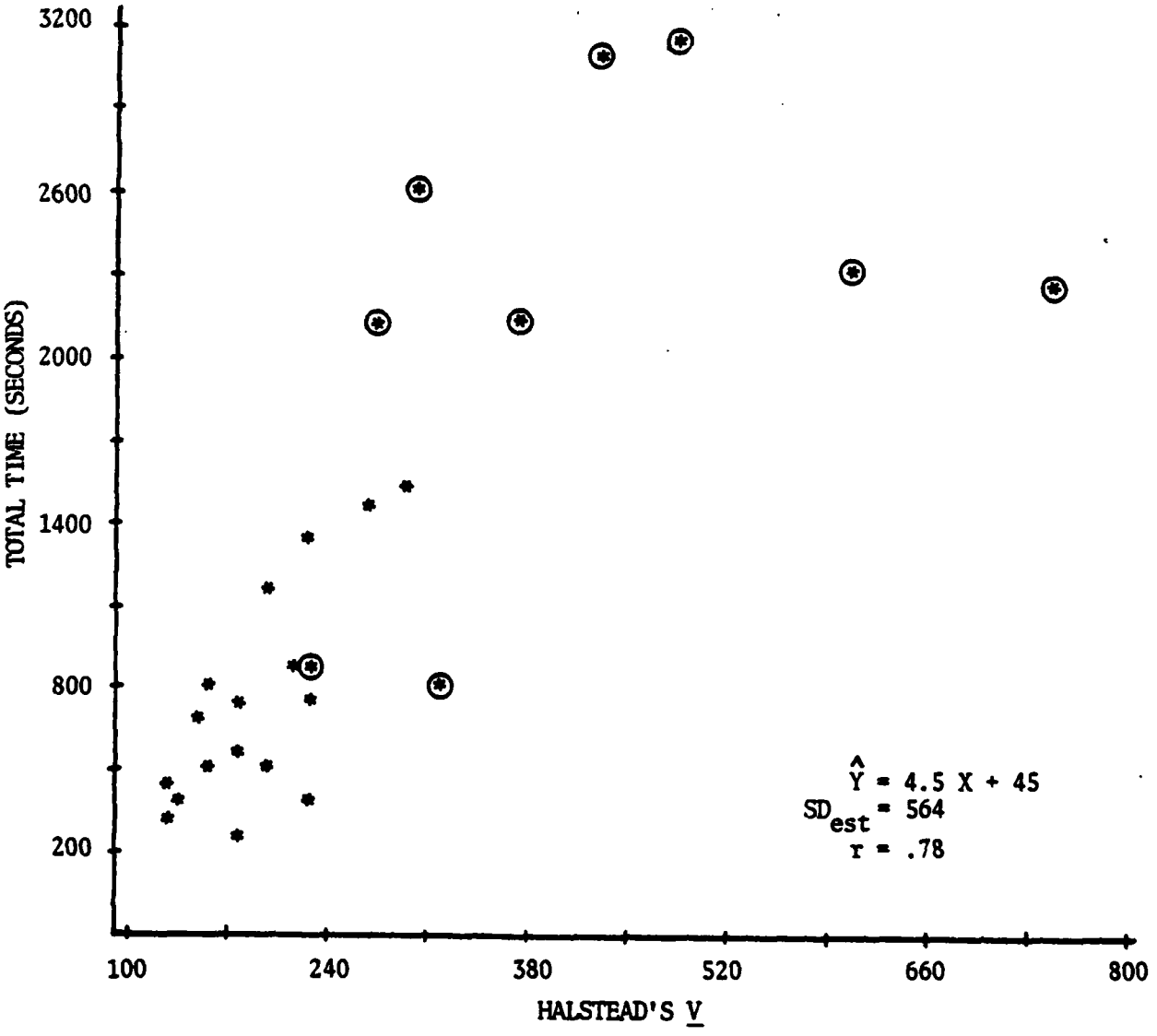


Figure 3. Scatterplot of Halstead's V with total time

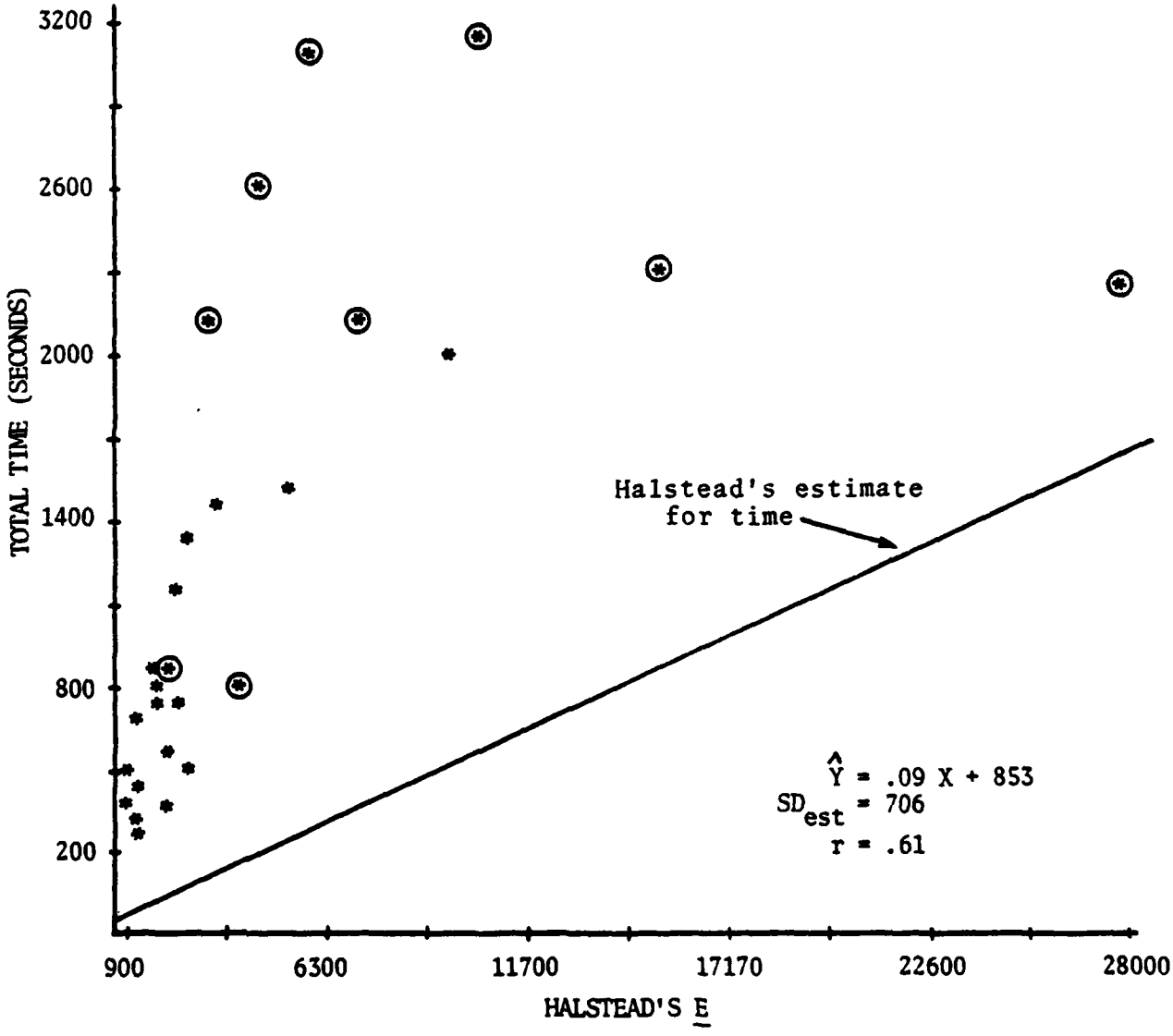


Figure 4. Scatterplot of Halstead's E with total time

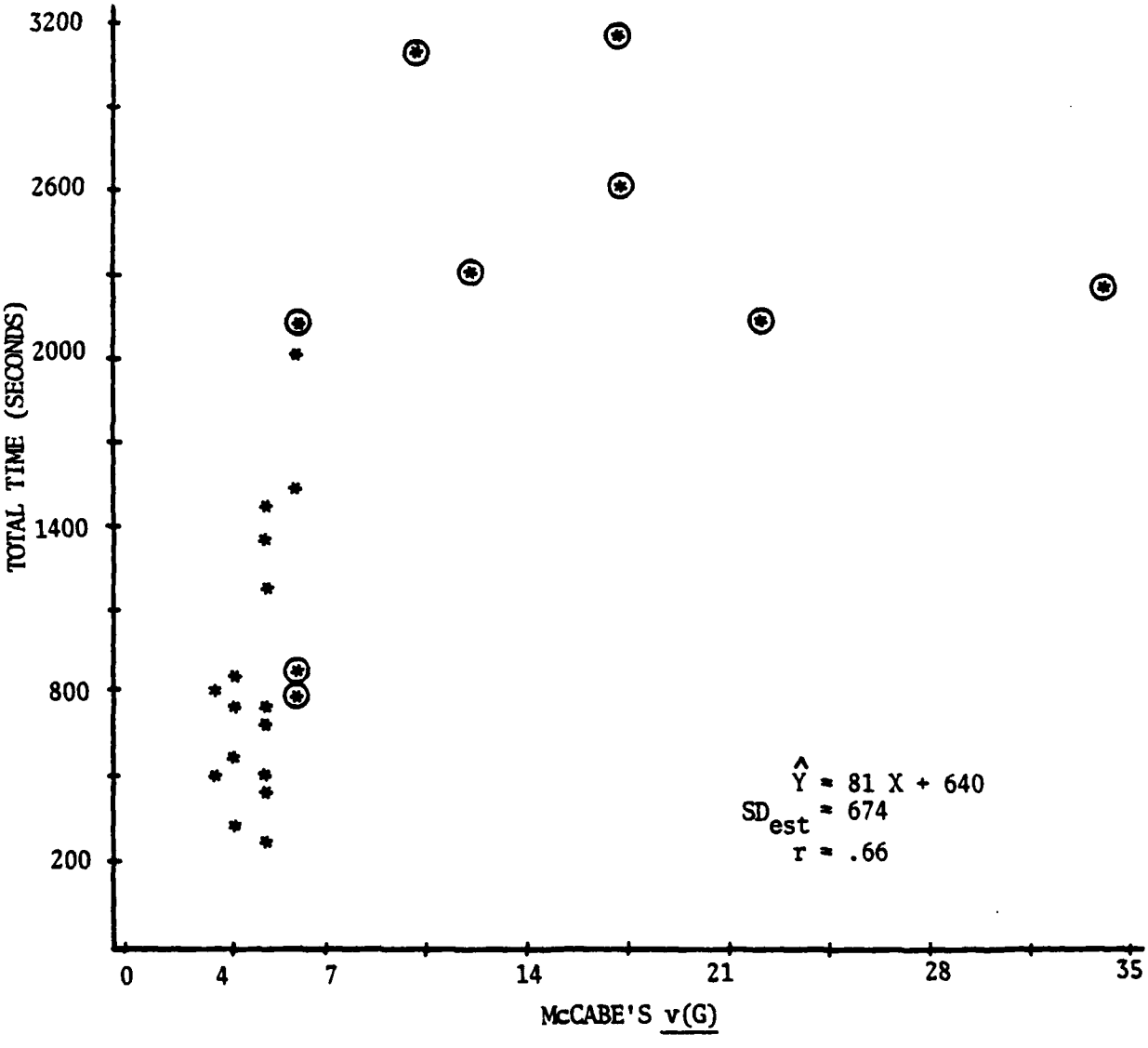


Figure 5. Scatterplot of McCabe's v(G) with total time

Total times during the three experimental tasks did not vary as a function of presentation order. Thus, it appears that the pretest provided sufficient experience to eliminate learning effects during the experimental tasks. Pretest time was a moderate predictor of time for the experimental tasks ($r = .40$, $p \leq .05$), suggesting the influence of individual differences among participants.

Analyses of the programs submitted by the participants yielded counts of the total number of actions to add, delete, change or list a program. (The renumber command was not used by the participants). The number of calls to the editor for the "ADD", "CHANGE", "DELETE", and "LIST" functions were also tallied. The number of actions was greater than or equal to the number of calls in each session, since more than one statement could be added, deleted, changed, or listed during a call to a given editor command. For example, if a programmer added five statements at one time, it would increase the number of actions by five but would increase calls to the editor for "ADD" by only one.

Descriptive statistics for editor actions are presented by run in Table 4. Only about half of the participants required more than one run to successfully complete their program. Very few participants required more than two runs. After the first run, the number of lines added decreased substantially, while the number of lines changed or deleted remained fairly constant.

Both the number of actions taken and the number of calls were analyzed for each compilation. Actions and calls performed during the first run (original entry of the program) were significant predictors of the total time to complete the experimental task (Table 5). That is, those programmers who manipulated the program listing more frequently (i.e., added, deleted, changed or listed) prior to the first submission of the program had longer total completion times. Actions taken during the second, third, and fourth submissions were few in number and not well correlated with performance.

Table 6 shows the correlations between the various actions and the Halstead metrics for each program. No differences were observed in the ability of these two metrics to predict the input and change or total editor actions taken by the participants.

No predictions about performance could be made from answers to the questionnaire. This inability probably resulted from the small sample size in this experiment ($n = 9$).

Table 4

Descriptive Statistics for Editor Actions by Run

Action	Mean	Standard Deviation	No. of Participants
ADDs			
Run 1	21.2	7.3	27
Run 2	13.5	16.2	4
Run 3	2.2	1.5	4
Run 4	0	0	0
CHANGEs			
Run 1	2.8	2.8	21
Run 2	2.2	1.4	13
Run 3	2.2	1.0	4
Run 4	1.0	0.0	2
DELETEs			
Run 1	3.0	2.2	10
Run 2	3.0	0.0	2
Run 3	2.0	0.0	1
Run 4	1.0	0.0	1
LISTs			
Run 1	42.2	47.9	24
Run 2	27.5	17.9	13
Run 3	33.7	35.9	3
Run 4	12.5	13.4	2

Table 5

Correlations of Actions with Total Time to Complete Program Successfully

Actions	For Run 1	For All Runs
Number of actions:		
ADDs	.71***	.46**
CHANGEs	.71***	.73***
DELETEs	.70***	.68***
LISTs	.74***	.74***
Total	.77***	.75***
Number of editor calls:		
ADDs	.71***	.77***
CHANGEs	.68***	.67***
DELETEs	.64***	.57***
LISTs	.78***	.73***
Total	.80***	.79***

Note: $n = 27$
**$p \leq .01$
***$p \leq .001$

Table 6

Correlations among Actions and Complexity Metrics

Actions	Halstead's Volume	Halstead's Effort
ADD	.63***	.68***
CHANGE	.49**	.39*
DELETE	.45**	.38*
LIST	.60***	.51**
Input and Change Actions (ADD, CHANGE, or DELETE):	.68***	.68***
Total Actions (ADD, CHANGE, DELETE, or LIST):	.64***	.57***

Note: $n = 27$
*$p \leq .05$
**$p \leq .01$
***$p \leq .001$

DISCUSSION

In this experiment, Halstead's volume and effort metrics and McCabe's cyclomatic number were better predictors of performance time than was the number of statements in the program. A previous experiment in this research program demonstrated moderate to good relationships between debugging performance and the Halstead and McCabe metrics (Curtis, Sheppard, & Milliman, 1979). However, in that experiment the metrics were computed on the prototype programs submitted to the participants. Since the experimental task in this experiment was to construct a program, it was possible to compute the metrics on the individual's own programs. The higher correlation of performance time with the Halstead and McCabe metrics in this experiment indicates that these metrics are better predictors of the psychological complexity of a program than is the number of statements in the program. Differences in the prediction of performance between the McCabe and Halstead metrics have not been consistent over the experiments in this research program. However, as we improved our experimental techniques and reduced the individual difference variation, the strongest results were obtained more consistently for the Halstead metrics.

The Halstead metric for the time required to generate a program underestimated the times required in this experiment. This result is not surprising since it is doubtful that dividing Halstead's effort metric by the Stroud number (18 mental discriminations per second) is related to the actual processes involved in constructing a program. The Stroud number is related to perceptual discrimination of simple stimuli (e.g., critical fusion frequency in integrating separate slides into a motion picture). Constructing programs requires more complex cognitive processing, and the Stroud number is probably inappropriate as a measure of this phenomenon. Further, any measure of mental processing time will differ both among people and in the same person over time, and would have to be recalibrated for each prediction. Thus, Halstead's time metric may correlate with programming time, but it is not an absolute measure of the phenomena.

At the level of the individual programmer, familiarity with relevant programming concepts is important for program construction. Program 1 required the matching of alphabetic strings and took over twice as much time as the other two programs. Program 2 required summing some negative and positive values, and Program 3 required finding the maximum and minimum of a set of numbers. The latter two are elementary operations that are performed often by programmers. These algorithms were probably more familiar to the nine participants than was the matching of strings.

Similar results for differences among algorithms were found in other experiments in this research program (Sheppard et al., 1979).

A frequent error for Program 1 involved comparing the masters to the list items instead of the reverse procedure. Some programmers did not initially perceive that the two methods of comparison produced different results. Thus they had to recode the algorithm to do the job correctly. The specifications for this part of Program 1, "CHECK FOR MASTERS", were not as complete and detailed as the specifications presented for Programs 2 and 3. It is therefore not surprising that a great deal more variation occurred in the performance times for Program 1. Informal analyses indicated that when an algorithm was insufficiently specified participants produced a variety of solutions, thus introducing greater variability into the construction process. In order to reduce variability in programmer performance, clear, detailed specifications for coding appear to be essential.

No differences were found among the methods of documentation presented along with the natural language descriptions of the problems. Informal conversations with the participants indicated that the additional documentation was often ignored. The problems were simple enough that most participants worked from the English description of the task. It is expected that in problems larger than those presented here the additional documentation would be useful. Predicting the relative merits of the documentation formats is not possible from this experiment.

A considerable learning effect took place during the pretest. The experimenter participated actively during this task, answering questions on the use of the microcomputer, the editor, and the Fortran compiler. Although the pretest problem was elementary (compute the average of a file of numbers), the time and number of tries required to execute this program were greater than for the experimental tasks. Learning to use the microcomputer and the editor during this pretest seemed to be adequate to eliminate the effects of learning during the experimental tasks themselves.

Averaged across all conditions, the first submission of the programs consumed 86% of the total time to completion (18.4 of the 21.3 minutes). Thus the actions (e.g., ADDS) during the first submission were excellent predictors of the total time to complete the program.

It is not surprising that those programmers who entered the largest number of statements had the longest completion

times for the tasks. It is interesting to note, however, that they also had the largest number of changes, deletes, and lists. Apparently those programmers who select more lengthy methods of solving problems increase their cognitive load and need greater numbers of adjustments (changes and deletes) and greater help in verifying the current state of the program listing (lists).

The number of calls to the editor for a command type appeared to predict total time as well as the number of statements. Clearly this result might be quite different with larger, more complex programs. If these data were to be validated for larger programs in other experiments, however, it would suggest that measuring the number of editor calls on the first submission or first few submissions might provide a project manager some information concerning overall performance.

The practical implications of the results presented here are substantial. All major software cost estimation models are driven from an estimate of the number of lines of code in the delivered product (Data Analysis Center for Software, 1979). The data presented here, however, argue that at the modular level, lines of code is a poor measure of the time required to construct a program. Putnam (1980) would suggest that large variances in these relationships would probably decrease substantially when data are aggregated at the system level. Nevertheless, in the data presented here the Halstead and McCabe metrics were significantly better predictors of performance time than was the number of statements. Once their values can be estimated, these complexity metrics may provide managers with better estimates of the resources and costs involved in later stages of the development project. Data from this experiment also suggest that the validity of these estimates may depend on whether sound principles of software engineering are observed in developing a system. Projects on which modern programming practices are enforced appear to experience more predictable results than projects without such discipline (Milliman & Curtis, 1979). This improved prediction results from the partial elimination of individual differences in performance achieved through enforcement of programming standards. Some refinement remains in developing software complexity metrics into a management information tool (Curtis, 1980), but their potential is evident.

ACKNOWLEDGEMENTS

We appreciate the assistance provided us by Dr. Tom Love in planning the research reported here, of Dr. Elizabeth Kruesi in examining the data, of Dr. John O'Hare in commenting on the manuscript, and of Beverly Day in preparing experimental materials and reports. We also appreciate the assistance of Lou Oliver in obtaining participants. The software package which performs the experimental data collection was developed under General Electric Independent Research and Development project 79D6A02, "Software Life Cycle Management Strategies".

REFERENCES

Boehm, B.W. Software and its impact: A quantitative assessment. Datamation, 1973, 19(5), 48-59.

Boehm, B.W., Brown, J.R., & Lipow, M. Quantitative evaluation of software quality. In Proceedings of the Second International Conference on Software Engineering. New York: IEEE, 1976, 592-605.

Boies, S. J., & Gould, J.D. Syntactic errors in computer programming. Human Factors, 1974, 16, 253-257.

Campbell, D., & Stanley, J.C. Experimental and quasi-experimental designs for research. Chicago: Rand-McNally, 1966.

Carlson, W.E., & DeRoze, B. Defense system software research and development plan. Arlington, VA: Defense Advanced Research Project Agency, September 1977.

Collins, A.M., & Quillian, M.R. Retrieval time from semantic memory. Journal of Verbal Learning and Verbal Behavior, 1969, 8, 240-248.

Curtis, B. In search of software complexity. In Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost. New York: IEEE, 1980, 95-106.

Curtis, B., Sheppard, S.B., & Milliman, P. Third time charm: Stronger prediction of programmer performance by software complexity metrics. In Proceedings of the Fourth International Conference on Software Engineering. New York: IEEE, 1979.

Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A., & Love, T. Predicting performance on software maintenance tasks with the Halstead and McCabe metrics. IEEE Transactions on Software Engineering, 1979, 5, 95-104.

Data Analysis Center for Software. Quantitative Software Models (SRR-1). Griffiss AFB, NY: Rome Air Development Center, 1979.

Department of Defense. Directive 5000.29, Management of Computer Resources in Major Defense Systems. 26 April 1976.

DeRoze, B. Software research and development technology in the Department of Defense. Paper presented at the AIIE Conference on Software, Washington, D.C., December 1977.

Dunsmore, H.E., & Gannon, J.D. The effect of individual programmer tendencies on the program construction process. College Park, Md: University of Maryland, Department of Computer Science, 1978.

Elshoff, J.L. A review of software measurement studies at General Motors Research Laboratories. In Proceedings of the Second Annual Software Life Cycle Management Workshop. New York: IEEE, 1978, 172-173.

Fitzsimmons, A.B., & Love, L.T. A review and evaluation of software science. ACM Computing Surveys, 1978, 10, 3-18.

Halstead, M.H. Elements of software science. New York: Elsevier, 1977.

Jackson, M.A. Principles of program design. London: Academic Press, 1975.

Jones, C. A survey of programming design and specification techniques. In Proceedings of the IEEE Conference on Specifications of Reliable Software. New York: IEEE, 1979, 91-103.

Kintsch, W., & van Dijk, T.A. Toward a model of text comprehension and production. Psychological Review, 1978, 85, 363-394.

Knuth, D.E. The art of computer programming: Fundamental algorithms. Reading, MA: Addison-Wesley, 1973.

Love, L.T. Relating individual differences in computer programming performance to human information processing abilities. (Doctoral dissertation, University of Washington) Dissertation Abstracts International, 1977, 38, 143b.

Lucas, H.C., & Kaplan, R.B. A structured programming experiment. The Computer Journal, 1974, 19, 136-138.

McCabe, T.J. A complexity measure. IEEE Transactions on Software Engineering, 1976, 2, 308-320.

McCall, J.A., Richards, P.K., & Walters, G.F. Factors in software quality (Tech. Rep. 77CIS02). Sunnyvale, CA: Command and Information Systems, General Electric, 1977.

McClure, C.L. Reducing COBOL complexity through structured programming. New York: Van Nostrand, 1978.

Miller, L.A., & Thomas, J.C. Behavioral issues in the use of interactive systems. Yorktown Heights, NY: IBM, 1976.

Milliman, P. & Curtis, B. An evaluation of modern programming practices in an aerospace environment. In Proceedings of the Third Digital Avionics Systems Conference. New York: IEEE, 1979.

Myers, G.J. A controlled experiment in program testing and code walkthroughs/inspections. Communications of the ACM, 1978, 21, 760-768.

Newsted, P.R. Fortran program comprehension as a function of documentation. Program Information Abstracts: Second Annual Computer Science Conference. Detroit, MI: 1974, 25.

Newsted, P.R. Flowchart-free approach to documentation. Journal of Systems Management, 1979, 30 (4), 18-21.

Ottenstein, K.J. A program to count operators and operands for ANSI-Fortran modules (Tech. Rep. 196). West Lafayette, IN: Purdue University, Computer Science Department, 1976.

Paivio, A. Imagery and verbal processes. New York: Holt, Rinehart & Winston, 1971.

Putnam, L.H. Software costing and life cycle control. In Proceedings of the Workshop on Quantitative Software Models for Reliability, Complexity, and Cost. New York: IEEE, 1980, 20-31.

Ramsey, H.R., Atwood, M.E., & Van Doren, J.R. A comparative study of flowcharts and program design languages for the detailed procedural specification of computer programs (Tech. Rep. #SAI-78-078-DEN). Englewood, CO: Science Applications Inc., 1978.

Sackman, H., Erickson, W.J., & Grant, E.E. Exploratory experimental studies comparing on-line and off-line programming performance. Communications of the ACM, 1968, 11, 3-11.

Sheppard, S.B., Curtis, B., Milliman, P., & Love, T. Modern coding practices and programmer performance. Computer, 1979, 12(12), 41-49.

Shneiderman, B. Software psychology: Human factors in computer and information systems. Cambridge, MA: Winthrop Press, 1980.

Shneiderman B., & Mayer, B.R. Syntactic/semantic interaction in programmer behavior: A model and experimental results. International Journal of Computer and Information Sciences, 1979, 8, 219-237.

Shneiderman, B., Mayer, B.R., McKay, D., & Heller, P. Experimental investigations on the utility of flowcharts in programming. Communications of the ACM, 1977, 20, 373-381.

Sime, M.E., Arblaster, A.T., & Green, T.R.G. Structuring the programmer's task. Journal of Occupational Psychology, 1977, 50, 205-216.

Stay, J.F. HIPO and integrated program design. IBM Systems Journal, 1976, 15, 143-154.

Stroud, J.M. The fine structure of psychological time. Annals of the New York Academy of Sciences, 1967, 623-631.

The military software market (Rep. 427). New York: Frost & Sullivan, 1977.

Wirth, N. Program development by stepwise refinement. Communications of the ACM, 1971, 14, 221-227.

Wright, P., & Reid, F. Written information: Some alternatives to prose for expressing the outcomes of complex contingencies. Journal of Applied Psychology, 1973, 57, 160-166.

Youngs, E.A. Human errors in programming. International Journal of Man-Machine Studies, 1974, 6, 361-376.

APPENDIX A

INSTRUCTIONS TO PARTICIPANTS

Instructions to Participants

Hello,

Today, we are asking you to participate in an experiment we hope will be both entertaining and challenging. This study is being sponsored by GE and the Office of Naval Research to examine the process of writing computer programs. To accomplish this, we will give you several different problems and ask you to implement them. Our purpose is not to evaluate computer programmers. Your performance on a program will be compared only to your performance on other programs, and no form of competition is involved. We hope you will assist us in what we believe is important research in software engineering. However, your involvement is voluntary and you are free to withdraw from participation at any time. All programs and papers that you will be handed are carefully numbered so it is not necessary for you to put your name on any of these. These numbers are solely for the purpose of identifying different problems and cannot be used to identify you as an individual. Your work will remain completely anonymous and data collected in this study will be used for research purposes only.

For each task, you will be given a problem description and the format and contents of the input files. Your job is to write the program to perform the requested function. Please make an attempt to do all you work on the computer, but if you wish to write notes down, please do so on the paper provided for each problem.

Directions for using the computer system are found on the following pages. You may ask any questions of the experimenter during the session: however, some questions he may require you to resolve yourself. The first problem will be a short introductory problem, familiarizing you with the computer and the types of problems you will receive. Upon completion of this problem, the computer will then lead you through the 3 problems of the experiment. Each problem will have an associated group of papers with instructions and documentation. Please turn to the appropriate set when told to do so by the computer. When you have completed all three experimental programs the computer will present a questionnaire about your past programming experience and then you are free to leave, but please do not discuss any of the programs you worked on with anyone else until after we have completed all experimental sessions. We request this of you only to insure that our results are valid.

If there are any questions, please ask them at this time. If you are willing to participate please sign your name on the line below indicating that you have read and understood these instructions.

_____

Signature

APPENDIX B

NOTES ON USING THE SYSTEM AND THE FORTRAN COMPILER

COMPUTER SYSTEM INSTRUCTIONS

When you first sit at the computer terminal, it will be in the command mode. The following prompt line will appear at the top of the screen:

ENTER COMMAND (E-EDIT, R-RUN, Q-QUIT)

A typical scenario might be to enter the editor (E), write a program, exit the editor and then run the program (R).

E: If you wish to enter some lines or make corrections to your program, type E and press the return key. This will put you in the editor portion of the program. A detailed description of the editor can be found on the next page.

R: If you wish to submit your program to compile and run, type R and press the return key. Call the supervisor, who will compile and run your program, during which time you may take a short break. If the compile is not successful, the supervisor will give you a printed listing of your compile errors. If the compile is successful, you will be given the run output. The supervisor will tell you whether the output is correct.

Q: The Quit command is to be used only when "all else fails", and it must be entered by the supervisor. If you feel you cannot continue with a specific program or the experiment, tell the supervisor and he will assist you in terminating the session.

EDITOR

Upon entry to the EDITOR program you will be given the following prompt:

```
EDIT COMMAND: (R-RENUMBER, L-LIST, D-DELETE, A-ADD,
               C-CHANGE, F-FINISH)
```

The editor is line-number oriented.

These numbers are assigned by you when you enter the A (ADD) command. The L (LOOK) and D (DELETE) commands reference existing lines and the R (RENUMBER) command establishes a new numbering scheme for the program. Line numbers are allowed in the following format:

> 0 to 3 digits optionally followed by a period and 0 to 3 digits. At least one character must appear. Each number is a fraction which can range from 000.001 to 999.999 for ADD, RENUMBER, and CHANGE, and 000.000 to 999.999 for LISTs and DELETEs.

Examples of invalid line numbers:

⌀ (blank)	• need at least 1 digit
X1	• X is not a digit or a "."
..01	• only one "." is allowed
.0001	• more than 3 digits to right of decimal
1000	• more than 3 digits to left of decimal

NOTE: If you wish, you may enter the line information immediately after the command letter instead of pressing return and being prompted.

R: To renumber the file, type R followed by a carriage return. You will be prompted to specify the First line number and, optionally, an Increment:

```
RENUMBER COMMAND: FIRST LINE[/INCREMENT]
```

The default Increment is 1. Neither the First line number nor the Increment may be 0. Neither number may be larger than 999.999

Example:

```
RENUMBER COMMAND: FIRST LINE[/INCREMENT] 20/10
```

```
program before renumber:  1        line 1
                          1.001    line 2
                          1.002    line 3
                        103        line 4
                        ----------------
program after renumber:   20       line 1
                          30       line 2
                          40       line 3
                          50       line 4
```

Examples of valid Renumber commands

3/4	Numbers programs at 3, 7, 11 ... until end
900	Numbers program at 900, 901 902, ... until end
.001/1	Numbers program at .001, 1.001 ... until end

Examples of invalid Renumber commands

/3	• must have a starting line
0	• cannot have line number or Increment = 0
/1/2	• only 1 slash is allowed to separate number and Increment
900/100	• illegal if there is more than 1 line because the second line will be greater than 999.999

L: To list lines in your program, type L. You will be prompted to state the first line you want to look at, followed optionally by a slash and the last line you want to look at.

LIST COMMAND: FIRST LINE[/LAST LINE]

If you only type the first line number, only that line is printed. If you enter both line numbers, the lines that fall within that range will be printed.

It is not an error if the first or the last line number entered do not have corresponding lines in the program. If there are not lines in the specified range a message will be printed.

Examples of valid List commands:

0/999.999	• This lists all the lines in the program. 0 is valid in look and can be used when you don't know the first line number.
99.03	• This lists only line number

099.030.

1/1	• This lists only line 001.000.

Examples of invalid commands:

/3	• You must have a first number
0 or 00	• There can be no line numbered 0 so no line will be found and listed. This command is not invalid, it is just meaningless.
/	• There must be a first number, and if there is a slash, there must also be a second number.
1000	• Line number is too large.

D: To Delete lines in your program, type D. You will be prompted to state the first line you wish to delete, follow optionally by a slash and the last line you want to delete.

DELETE COMMAND: FIRST LINE[/LAST LINE]

If you only enter the First number, then only that line is deleted. If you enter both line numbers, all the lines that fall within that range are deleted. It is not an error if the First or the Last line number entered do not have corresponding lines in the program. If there are no lines in the specified range a message will be printed.

Examples of valid Delete commands:

0/999.999	• Deletes all lines in program. 0 is valid for start of range and serves as a useful tool to include the first line number.
99.03	• Deletes only line 099.030 if it exists.
1/1	• Deletes only line 001.000.

Examples of invalid commands:

/3	• You must have a first number.
0 or 0/0	• There can be no line numbered 0 so no line will be found and deleted. This command is not invalid, it is just meaningless.
0/	• If there is a slash there must also be a second number.
1000	• Number is too large.

A: To Add lines to the program, type A. You will be prompted to enter the First line number you wish to Add,

followed optionally by a slash and the Increment.

ADD COMMAND: FIRST LINE[/INCREMENT]

If you omit the slash and Increment then the Increment defaults to 001.000. The First line number must be already exist. For instance if your program appears as follows,

```
1       line 1
2       line 2
```

you may Add any line but 1 and 2. This includes the range from .001 to .999, 1.001 to 1.999, and 2.001 to 999.999.

You will be prompted with the First line number. You may enter up to 72 characters. After you enter a line and press Return it will be included in your program at the place its number indicates. You will then be prompted for the next line number, which is determined by adding the Increment to the line number of the line you just finished entering. If the next line number is equal to or greater than the number of an already existing line in the program, the Add will be terminated. All the lines you have already entered will remain in the program and will not be affected by the termination. When you finish entering the lines you want, enter two slashes (//) as the first characters of a line and press return. This will return you the EDIT COMMAND level.

The following examples refer to the example shown above:

1	• Rejected because line 1 already exists.
1.01	• This will allow one line to be added: then, since 2.01 (1.01 + 1) is greater than or equal to 2, the Add will be terminated.
0	• Rejected because 0 can not be a line number.
5	• You may enter lines from 5, 6, 7, ... 999.
/.5	• Rejected because there is no line number.
1.4/.1	• This will allow you to add up to 6 lines 1.4, 1.5, 1.6, 1.7, 1.8, 1.9 before it terminates because it passes or equals 2.
5/	• Rejected because if there is a slash an increment must follow.

Sample session:

ADD COMMAND: FIRST LINE[/INCREMENT] 1.3/.2

```
1.3  you type something here
1.5  you also type something here
1.7  //
```

The program would now look like this:

```
1    line 1
1.3  you type something here
1.5  you also type something here
2    line 2
```

C: To change an existing line, type C. You will be prompted to enter the line number of the line you wish to change:

CHANGE COMMAND: LINE NUMBER

The system will then type the line as it currently is. Enter the line you want, then press return. If you decide you really didn't want to make a change, enter two slashes (//) followed by a return. The system will leave the line as it was.

Given the sample program;

```
1       LINE 1
2       LINE 2
```

CHANGE COMMAND: LINE NUMBER 1 would perform the following

```
1       LINE 1
1
```

You enter the new line contents and press return. (E.G. NEW LINE). The program would now look like this:

```
1       NEW LINE
2       LINE 2
```

If you had entered // the program would not have been changed.

The following examples refer to the sample program given above:

2	• Allows you to change line 002.000.
3	• Rejected because there is no line 003.000.
1/2	• Rejected because only one line is allowed to be Changed at a time

0 • No line 0 exists or is allowed to exist.

F: When you wish to return to the main command mode (Usually to submit your program for a run) enter F followed by return.

Notes on typing on the computer:

The following rules apply to your interaction with the computer before you press the return key.

Erase characters - if you wish to erase the last character you typed, press the DELETE key. If you wish to erase more, keep pressing the delete key.

Erase a line - If you wish to start the present line over, press the LINE DEL key and it will erase all characters you have just typed. You may then re-enter your command or program line.

After you have pressed return you will have to live with the consequence of your actions.

DO NOT USE TAB characters. This program is not set up to handle them.

FORTRAN NOTES

The FORTRAN you will be using is STANDARD FORTRAN IV with the few differences noted below:

1. Opening files for input should be done in the following manner:

```
CALL OPEN (6, 'filename', 0)
```

   The file name must be followed by four blanks and must be in single quotes. If the input file is ONRD001, the statement would be:

```
CALL OPEN (6, 'ONRD001bbbb' 0)
```

   It is not necessary to close the file.

2. All read statements use 6 as the device unit number; e.g.,

```
          READ (6, 100) A, B, C
  100     FORMAT (3F10.3)
```

3. All output statements use device number 2; e.g.,

```
          WRITE (2,300) I, A
  300     FORMAT (10X, 'I=', I5, 'A=', F10.3)
```

4. Only variables are allowed in WRITE lists. No expressions or constants are allowed

   acceptable:

```
WRITE (2, 2) A, B, C
WRITE (2, 400) (A(I), I = 1, 10)
WRITE (2, 346)
```

   not acceptable:

```
WRITE (2,2) A, B, 3            3 is a constant
WRITE (2, 400) A+1             A+1 is an expression
WRITE (2, 195) I-J, B, D       I-J is an expression
```

5. End of file or ERROR branches may be specified in READ or WRITE statements using the ERR= and END= options; e.g.,

```
 10  READ (6, 2, END= 100, ERR=200) A, B, C
  2  FORMAT (3F10.2)
       •
       •
```

```
      GO TO 10
100   WRITE (2, 3)
  3   FORMAT (' END OF FILE')
      GO TO 999
200   WRITE (2, 4)
  4   FORMAT (' ERROR IN READ')
999   STOP
      END
```

6. NO COMPLEX Data Types are allowed.

7. DO loops will always be executed at least once.

8. Hollerith constants can be represented by the NHString format or enclosed in single quotes.

   Examples:

A = 'A' or 1HA	A has been previously defined to be type logical by the programmer
I = 'CE' or 2HCE	I has been previously defined to be type Integer by the programmer
R = 'JXYZ' or 4HJXYZ	R has been previously defined to be type real by the programmer

   If fewer characters than the allotted space are given, the Hollerith values are left justified with trailing blanks.

9. Mixed Mode expressions and assignments are allowed, and conversions are done automatically.

10. The specification statements must appear in the following order:

    A. PROGRAM or SUBROUTINE or FUNCTION or BLOCK DATA
    B. Type or EXTERNAL or DIMENSION
    C. COMMON
    D. EQUIVALENCE
    E. DATA
    F. Statement Functions

APPENDIX C

PRETEST PROBLEM

FIRST PROBLEM: PRACTICE

Find the average of the numbers in file 'ONRDØØPØØØØ'. There is one number per record, in the format (I5). There are at least two and at most 5Ø records in the data file. Your one-line output should contain the average in the following format: ('Ø', F5.1).

Sample Input:

3
7
5
<EOF>

Sample Output:

5.Ø

APPENDIX D

EXAMPLES OF EXPERIMENTAL PROBLEMS

Problem 1: Shown with PDL and tree structure

Problem 2: Shown with PDL only

Problem 3: Shown with tree structure only

PROBLEM 1

Each record in file 'ONRD##Ø1ßßßß' contains three alphabetic characters in the format (1X, 3A1). The first record has master characters to which all other characters must be compared. Your task is to count the number of records containing all three of the master characters in any order. The input format is (1X, 3A1). There is at least one but not more than 50 records on the file. Your output format should be: ('ß', I3).

Sample Input:

ABC
CBA
AAC
ABC
CCB
ABX
≪EOF≫

Sample Output:

2

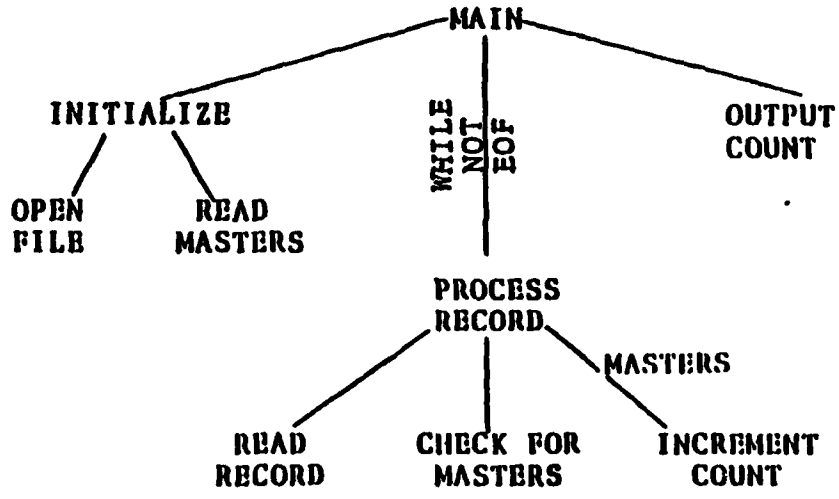
Program Process

Main:

Initialize
  Open file
  Read masters

While not EOF process record
  Read record
  Check for masters
  If masters, increment count

Output count



PROBLEM 2.

A series of positive and negative values are on file 'ONRDØØ2ØØØØ' in format (F6.2). Your task is to output the sum of the negative values, the sum of the positive values, and the total. The output format is ('Ø', 3F9.2). There is at least one but not more than 50 records on the file.

Sample Input:

-1.01
123.56
0.53
-12.12
EOF

Sample Output:

-13.13 124.09 110.96

Program Process

Main:

Initialize
  Open file
  Clear sums

While not EOF process record
  Read value
  Check sign
    If <0 add to neg. sum
       ≥0 add to pos. sum

Compute total sum

Output sums

PROBLEM 3.

Find the minimum and the maximum of the numbers in 'ONRD##3####'. Input records are in the format (I5). There is at least one and at most 50 items on the file with no items greater than 999 or less than -999. Output should contain the minimum, then the maximum in format ('Ø', 2I5).

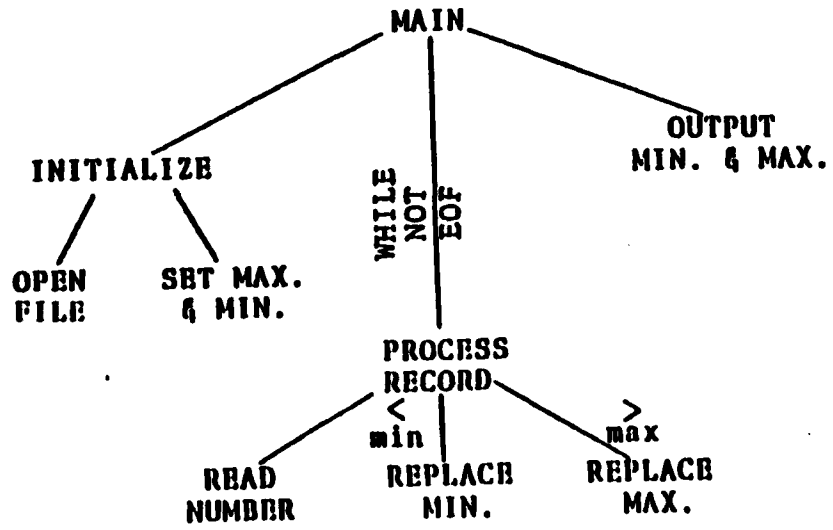
Sample Input:

```
-32
356
  1
  Ø
426
 1Ø
≪EOF≫
```

Sample Output:

```
-32   426
```


Program Process



APPENDIX E

QUESTIONNAIRE

```
***WE WOULD LIKE YOU TO ANSWER THE FOLLOWING QUESTIONS FOR***
***OUR RESEARCH PURPOSES:                                  ***
***                                                        ***
***PLEASE ANSWER EACH QUESTION ON A SINGLE LINE            ***
***AFTER TYPING THE ANSWER TO A QUESTION, PRESS RETURN.    ***
***                                                        ***
***IF THE ANSWER TO A QUESTION IS YES OR NO, YOU MAY       ***
***     ENTER Y OR N.                                      ***
***IF YOU CAN NOT ANSWER A QUESTION ENTER - DON'T KNOW     ***
***IF THE QUESTION IS NOT APPLICABLE ENTER - N/A           ***
***
HOW MANY YEARS HAVE YOU BEEN PROGRAMMING PROFESSIONALLY?
HOW MANY YEARS HAVE YOU BEEN PROGRAMMING FORTRAN PROFESSIONALLY?
HAS YOUR EXPERIENCE BEEN PRIMARILY WITH
        A. ENGINEERING
        B. STATISTICAL
        C. NON-NUMERICAL
        D. BUSINESS
        E. DATA BASE
        F. OTHER(PLEASE SPECIFY
HOW MANY LINES WERE IN YOUR LONGEST FORTRAN PROGRAM?
HOW MANY LINES WERE IN YOUR LONGEST NON-FORTRAN PROGRAM?
IN WHAT LANGUAGE WAS YOUR LONGEST NON-FORTRAN PROGRAM?
HAVE YOU USED:
        FORTRAN?
        FORTRAN 77?
        COBOL?
        PL/1?
        BASIC?
        PASCAL?
        APL?
        ALGOL?
        JOVIAL?
        ASSEMBLER?
        RPG?
        SNOBOL?
        LISP?
        OTHER(GIVE NAME)?
WHAT WAS THE FIRST LANGUAGE YOU LEARNED?
IN YOUR PROGRAMMING HAVE YOU USED:
        DO statement?
        arrays?
        CALL with parameters?
        COMMON?
        READ statement?
        PRINT statement?
        WRITE statement?
        FORMAT statement?
        'X' format specification?
        'A' format specification?
        'I' format specification?
        'F' format specification?
        continuation lines?
        'H' format specification?
        implicit data types?
        IF THEN ELSE (concept)?
        CREDITS in monetary transactions?
        DEBITS in monetary transactions?
        Financial transactions?
        TRIAL BALANCE computation?
```

GENERAL LEDGER accounting?
REAL notation(0.01)?
tax computation?
carriage control Hollerith?
2 dimension or more arrays?
using quotes to delimit strings in output formats?
IMPLICIT statement?
heap sorts?
stacks?
tree search?
NAMELIST statement?
'T' format specification?
interrupt handlers?
parsers?
lexical analyzers?
graphics drivers and handlers?
DATA statement?
conversion from alpha to string variables?
IF of more than 1 condition?
decimal to integer conversion?
percentile computation?
DO WHILE concept?
DO UNTIL concept?
weighting numbers or scores?
rounding numbers when don't have rounding function?
used an array reference as an index to another array?
finding maximum value in an array?
finding mean of values in an array?
printing titles on output?
computing frequencies of items?
running sums?
bubble sort?
implied DO?
equivalenced arrays?
string variables?
the binary equivalent of characters?
interactive debugger?
symbolic debugger?
TRACE mechanism?
Octal or Hex dumps?
double precision?
free field I/O?
matrix inversion?
pattern matching?
device drivers?
batch systems?
interactive systems?
list handling languages?

***

On the back of your instruction sheet we would like you to indicate any other particulars which you feel may have an effect on your performance. For instance, we would like to know if most of your work is in debugging systems or in design. Also, please indicate your reactions to the experiment and anything that you feel might help us to improve it. If you pursued the task in any special way we would like to know.

***

APPENDIX F

SOFTWARE EXPERIMENTATION LABORATORY

SOFTWARE EXPERIMENTATION LABORATORY

Experiments in software development depend very heavily upon the methods and tools used. Research has typically involved either field studies on large programs or experiments on small artifical problems. The experimental process requires strict controls over system response for accurate timing and consistent presentation of materials. All interactions of the subject with the computer must be recorded so that his actions can be reconstructed at a later time. For ease of debugging and modification the experimental system should be implemented using a high level language.

The Software Management Research group has established a software experimentation laboratory around a Northwest Microcomputer Systems NMS 85/P. This microcomputer uses both the UCSD PASCAL operating system and the CPM operating system which supports a Fortran compiler. The system has a CRT for display, a printer to allow hard copy generation and an interval timer to provide timing accurate to .01 seconds.

The PASCAL language was chosen for implementing the experimentation software for several reasons. It is a simple language with a unified concept, easy to learn, yet powerful. PASCAL allows the building of data structures, such as strings, records, and pointers, allowing the implementer to model a problem in a manner close to the real-world representation. These factors are expected to contribute to the ease of implementing, maintaining, and modifying the experimental software.

The experimental system established for the microprocessor laboratory is designed to take the user step by step through the experiment. It provides the environment for a pretest and the three experimental conditions and gives the user an online questionnaire.

The editor is designed to be easy to learn and use. Users see where they are and know which lines they are working with at any time. A structure is set up for using a random access file for edit lines. The edit lines are organized around a linked list structure, and the user references them by line numbers. The disadvantages of this approach are slow speed and heavy I/O usage, but in a single user environment this is not felt to be a problem. The

advantages are that any changes made are immediately updated to the edit file. (Since microprocessors do not tend to be as highly reliable as minicomputers and mainframes it was necessary to be ready for the unexpected). Another positive aspect is that the subject does not have to be concerned with buffering problems. As far as he knows, his file is essentially infinite; he does not have the difficulty of asking for previous or forward buffers.

The editor allows five commands to operate on a file - ADD, DELETE, LIST, RENUMBER, and CHANGE. There is only one file for each problem, so the computer does the file handling. Each line in the edit file is assigned a number by the user, which is referenced whenever an edit operation is performed. The ADD command allows the user to add 1 or more lines to any location not already existing as a line number. The DELETE and LIST commands allow a range of lines to be deleted and listed. The RENUMBER command allows the user to re-assign line numbers to the file based upon a starting line number and an increment. The CHANGE command allows the user to reenter a given line. There are no search or substitute commands. It was felt that these would add to the complexity of the editor and would not provide great utility given the small size of the programs to be constructed.

When users are ready to submit a job for compilation and execution they exit from the editor and request that their program be run. This is a one letter command, and the microcomputer handles the next steps. At present, the Fortran compile is performed under a different operating system on the same computer. The system is turned over to the experimenter who transfers the edit file to the CPM system to do a Fortran compile. If the compile is successful, the experimenter then runs the program. In either case, the experimenter transfers the results back to the PASCAL system and resumes the experimental program. If the experimenter indicates that the run is successful, the experimental control program proceeds to the next condition. If there is a compile error or run error, the program stays in the same edit file and allows the user to resolve the problem.

After completion of the pretest and the three experimental problems, the users are given a questionnaire on-line about their experience in software development (mostly programming). The results are stored on a file for future analysis.

This system opens the door to a wide variety of research efforts, allowing a greater degree of objectivity and

accuracy than is normally available for research in this field. The present experimental program will need only minor changes to provide the format of experiments on debugging and modification. The program will be modified to display an existing file rather than one that is created by the user. It is simple to add a FIND command to allow the user to search for relevant strings in the file.

With minor changes it is possible to perform psychological experimentation, reaction time studies, or even automated questionnaire presentation. Other enhancements would be the interfacing of this system with other systems. The NMS 85/P has an RC 232C interface so all that is required is to change the current submission module to handle the protocol of a different system. This would allow a programmer's workbench arrangement for the user, allowing an audit trail to be obtained from performance with different languages while presenting a uniform environment to the user.

OFFICE OF NAVAL RESEARCH

Code 455

TECHNICAL REPORTS DISTRIBUTION LIST

OSD

CDR Paul R. Chatelier
Office of the Deputy Under Secretary
of Defense
OUSDRE (E&LS)
Pentagon, Room 3D129
Washington, D.C. 20301

Department of the Navy

Director
Engineering Psychology Programs
Code 455
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217 (5 cys)

Director
Information Systems Program
Code 437
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Director
Physiology Program
Code 441
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Special Assistant for Marine
Corps Matters
Code 100M
Office of Naval Research
800 North Quincy Street
Arlington, VA 22217

Department of the Navy

Commanding Officer
ONR Branch Office
ATTN: Dr. J. Lester
Building 114, Section D
666 Summer Street
Boston, MA 02210

Commanding Officer
ONR Branch Office
ATTN: Dr. C. Davis
536 South Clark Street
Chicago, IL 60605

Commanding Officer
ONR Branch Office
ATTN: Dr. E. Gloye
1030 East Green Street
Pasadena, CA 91106

Office of Naval Research
Scientific Liaison Group
American Embassy, Room A-407
APO San Francisco, CA 96503

Human Factors Engineering Branch
Code 1226
Pacific Missile Test Center
Point Mugu, CA 93402

ONR Code 455, Technical Reports Distribution List

Department of the Navy

Director
Naval Research Laboratory
Technical Information Division
Code 2627
Washington, D.C. 20375 (6 cys)

Dr. Bruce Wald
Communications Sciences Division
Code 7500
Naval Research Laboratory
Washington, D.C. 20375

Dr. Robert G. Smith
Office of the Chief of Naval
Operations, OP987H
Personnel Logistics Plans
Washington, D.C. 20350

Naval Training Equipment Center
ATTN: Technical Library
Orlando, FL 32813

Human Factors Department
Code N215
Naval Training Equipment Center
Orlando, FL 32813

Dr. Alfred F. Smode
Training Analysis and Evaluation
Group
Naval Training Equipment Center
Code N-00T
Orlando, FL 32813

CDR R. Gibson
Bureau of Medicine & Surgery
Aerospace Psychology Branch
Code 513
Washington, D.C. 20372

Department of the Navy

Dr. Arthur Bachrach
Behavioral Sciences Department
Naval Medicial Research Institute
Bethesda, MD 20014

Dr. George Moeller
Human Factors Engineering Branch
Submarine Medical Research Lab
Naval Submarine Base
Groton, CT 06340

Head
Aerospace Psychology Department
Code L5
Naval Aerospace Medical Research Lab
Pensacola, FL 32508

Navy Personnel Research and
Development Center
Manned Systems Design, Code 311
San Diego, CA 92152

Navy Personnel Research and
Development Center
Code 305
San Diego, CA 92152

Navy Personnel Research and
Development Center
Management Support Department
Code 210
San Diego, CA 92151

CDR P. M. Curran
Code 604
Human Factors Engineering Division
Naval Air Development Center
Warminster, PA 18974

ONR Code 455, Technical Reports Distribution List

Department of the Navy

Dean of the Academic Departments
U.S. Naval Academy
Annapolis, MD 21402

Dr. Gary Poock
Operations Research Department
Naval Postgraduate School
Monterey, CA 93940

Dean of Research Administration
Naval Postgraduate School
Monterey, CA 93940

Mr. Warren Lewis
Human Engineering Branch
Code 8231
Naval Ocean Systems Center
San Diego, CA 92152

Dr. A.L. Slafkosky
Scientific Advisor
Commandant of the Marine Corps
Code RD-1
Washington, D.C. 20380

Mr. Arnold Rubinstein
Naval Material Command
NAVMAT 08D22
Washington, D.C. 20360

Commander
Naval Air Systems Command
Human Factors Programs
NAVAIR 340F
Washington, D.C. 20361

Department of the Navy

Commander
Naval Air Systems Command
Crew Station Design
NAVAIR 5313
Washington, D.C. 20361

Naval Sea Systems Command
Personnel & Training Analyses Office
NAVSEA 074C1
Washington, D.C. 20362

Commander
Naval Electronics Systems Command
Human Factors Engineering Branch
Code 4701
Washington, D.C. 20360

Human Factors Section
Systems Engineering Test Directorate
U.S. Naval Air Test Center
Patuxent River, MD 20670

Human Factor Engineering Branch
Naval Ship Research and Development
Center, Annapolis Division
Annapolis, MD 21402

LCDR W. Moroney
Code 55MP
Naval Postgraduate School
Monterey, CA 93940

Mr. Merlin Malehorn
Office of the Chief of Naval
Operations (OP 102)
Washington, D.C. 20350

ONR Code 455, Technical Reports Distribution List

Department of the Army

Mr. J. Barber
HQS, Department of the Army
DAPE-MBR
Washington, D.C. 20310

Dr. Joseph Zeidner
Technical Director
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Dr. Edgar M. Johnson
Organizations and Systems Research Laboratory
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Technical Director
U.S. Army Human Engineering Labs
Aberdeen Proving Ground, MD 21005

U.S. Army Aeromedical Research Lab
ATTN: CPT Gerald P. Krueger
Ft. Rucker, AL 36362

ARI Field Unit-USAREUR
ATTN: Library
C/O ODCSPER
HQ USAREUR & 7th Army
APO New York 09403

Department of the Air Force

U.S. Air Force Office of Scientific Research
Life Sciences Directorate, NL
Bolling Air Force Base
Washington, D.C. 20332

Mr. Donal A. Topmiller
Chief, Systems Engineering Branch
Human Engineering Division
USAF AMRL/HES
Wright-Patterson AFB, OH 45433

Air University Library
Maxwell Air Force Base, AL 36112

Dr. Gordon Eckstrand
AFHRL/ASM
Wright-Patterson AFB, OH 45433

Foreign Addressees

North East London Polytechnic
The Charles Myers Library
Livingstone Road
Stratford
London E15 2LJ
ENGLAND

Director, Human Factors Wing
Defense & Civil Institute of Environmental Medicine
Post Office Box 2000
Downsview, Ontario M3M 3B9
CANADA

ONR Code 455, Technical Reports Distribution List

Foreign Addressees

Dr. A.D. Baddeley
Director, Applied Psychology Unit
Medical Research Council
15 Chaucer Road
Cambridge, CB2, 2EF
ENGLAND

Other Government Agencies

Defense Documentation Center
Cameron Station, Bldg. 5
Alexandria, VA 22314 (12 cys)

Dr. Craig Fields
Director, Cybernetics Technology Office
Defense Advanced Research Projects Agency
1400 Wilson Blvd
Arlington, VA 22209

Dr. Stanley Deutsch
Office of Life Sciences
National Aeronautics and Space Administration
600 Independence Avenue
Washington, D.C. 20546

Dr. Gary McClelland
Institute of Behavioral Sciences
University of Colorado
Boulder, CO 80309

Human Resources Research Office
300 N. Washington Street
Alexandria, VA 22314

Other Government Agencies

Dr. Jesse Orlansky
Institute for Defense Analyses
400 Army-Navy Drive
Arlington, VA 22202

Dr. Robert G. Pachella
University of Michigan
Department of Psychology
Human Performance Center
330 Packard Road
Ann Arbor, MI 48104

Dr. Arthur I. Siegel
Applied Psychological Services, Inc.
404 East Lancaster Street
Wayne, PA 19087

Journal Supplement Abstract Service
American Psychological Association
1200 17th Street, N.W.
Washington, D.C. 20036 (3 cys)

Mr. Edward Connelly
Performance Measurement Associates, Inc.
131 Park Street, NW
Vienna, VA 22180

Dr. Edward R. Jones
Chief, Human Factors Engineering
McDonnell-Douglas Astronautics Company
St. Louis Division
Box 516
St. Louis, MO 63166